Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 387 (2007) 313–331
Theoretical Computer Science
www.elsevier.com/locate/tcs

Compressed data structures: Dictionaries and data-aware measures☆

Ankur Gupta[a], Wing-Kai Hon[b], Rahul Shah[c], Jeffrey Scott Vitter[d,*]

[a] *Department of Computer Science, Butler University, Indianapolis, IN 46208, USA*
[b] *Department of Computer Science, National Tsing Hsu University, Taiwan*
[c] *Department of Computer Science, Louisiana State University, LA 70803, USA*
[d] *Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-2066, USA*

Abstract

In this paper, we propose measures for compressed data structures, in which space usage is measured in a data-aware manner. In particular, we consider the fundamental *dictionary problem* on *set data*, where the task is to construct a data structure for representing a set S of n items out of a universe $U = \{0, \ldots, u-1\}$ and supporting various queries on S. We use a well-known data-aware measure for set data called *gap* to bound the space of our data structures.

We describe a novel dictionary structure that requires $gap + O(n \log(u/n)/\log n) + O(n \log\log(u/n))$ bits. Under the RAM model, our dictionary supports membership, rank, and predecessor queries in nearly optimal time, matching the time bound of Andersson and Thorup's predecessor structure [A. Andersson, M. Thorup, Tight(er) worst-case bounds on dynamic searching and priority queues, in: ACM Symposium on Theory of Computing, STOC, 2000], while simultaneously improving upon their space usage. We support select queries even faster in $O(\log\log n)$ time.

Our dictionary structure uses exactly *gap* bits in the leading term (i.e., the constant factor is 1) and answers queries in near-optimal time. When seen from the worst-case perspective, we present the first $O(n \log(u/n))$-bit dictionary structure that supports these queries in near-optimal time under the RAM model. We also build a dictionary which requires the same space and supports membership, select, and partial rank queries even more quickly in $O(\log\log n)$ time.

We go on to show that for many (real-world) datasets, data-aware methods lead to a worthwhile compression over combinatorial methods. To the best of our knowledge, these are the first results that achieve data-aware space usage and retain near-optimal time.

Keywords: Dictionary problem; Compressed; Gap encoding; Rank; Select; Predecessor; BSGAP

1. Introduction

The proliferation of data is a problem that is suffocating our abilities to manage information. Massive data sets from biological experiments, Internet routing information, sensor data, and audio/video devices require new methods

☆ Support was provided in part by the Army Research Office through grant DAAD20-03-1-0321 and by the National Science Foundation through research grant IIS-0415097.

* Corresponding address: Department of Computer Science, Duke University, 27708-0129 Durham, NC, USA.
E-mail addresses: agupta@cs.purdue.edu (A. Gupta), wkhon@cs.nthu.edu.tw (W.-K. Hon), rahul@cs.purdue.edu (R. Shah), jsv@cs.purdue.edu (J.S. Vitter).

0304-3975/$ - see front matter © 2007 Published by Elsevier B.V.
doi:10.1016/j.tcs.2007.07.042

for managing data. In many of these cases, the information content is relatively small compared to the size of the original data. We want to exploit the huge potential to save space in these cases. However, in many applications, data also needs to be indexed for fast query processing. The new trend of data structure design considers time and space efficiency together: The ultimate goal is to build structures that operate in the optimal (or nearly so) time bound, while requiring the minimum amount of space, tuned for the particular input data.

Ideally, the space required for a structure should be defined with respect to the *Kolmogorov complexity* of the data upon which the structure is built, as it is the space of the smallest program that can generate the input data. Unfortunately, it is undecidable for arbitrary input, making it an inconvenient measure for practical use. Thus, other measures of compressibility are used as a framework for data compression, like *entropy* for textual data.

One fundamental type of data is *set data*, which consist of a subset S of n items from a universe $U = \{0, \ldots, u-1\}$. Some specific examples include IP addresses, UPC barcodes, and ISBN numbers: set data also appear in inverted indexes for libraries and web pages, as well as results from scientific experiments. In many natural examples of set data, S is not a random subset of U and can be compressed. (For instance, consider a set S with a few tightly clustered items spread throughout U.)

In this paper, we use the *gap* measure [6] (described formally in Section 2.2), which has been used extensively as a reasonable space measure in the context of inverted indexes [23]. The gap measure counts the space required to encode the distances between successive items and is usually much less than the information-theoretic lower bound of $\lceil \log \binom{u}{n} \rceil \approx n \log(u/n)$ bits.[1] (This bound is known as the information-theoretic minimum because it is the minimum number of bits needed to differentiate the $\binom{u}{n}$ possible subsets of n items out of a universe of size u.) A *gap*-style encoding can be potentially much smaller than $\lceil \log \binom{u}{n} \rceil$ bits for many of the data sets above, since it exploits short distances between items.

We use these notions of compressibility to design *compressed data structures* that index the data in a succinct way and also allow fast access. In particular, we address the fundamental *dictionary problem*, where we design a data structure to represent a subset S that supports various queries on S. In this paper, we present compressed representations for both fully-indexable dictionaries (FID) and indexable dictionaries (ID), improving the space required by previous results while maintaining near-optimal query time. In particular, under the unit-cost RAM model, we develop a fully-indexable dictionary (FID) – a data structure supporting rank and select queries – of size $gap + O(n \log(u/n)/\log n) + O(n \log\log(u/n))$ bits, while supporting rank in time matching Andersson and Thorup's (nearly optimal) predecessor structure [1] and select even faster in $O(\log\log n)$ time. When $n \in o(u)$, our fully-indexable dictionary is asymptotically equal to *gap* space (with a constant of 1). This is important because, for most real-life data, $n \ll u$ and *gap* is significantly less than the worst-case information-theoretic minimum $\lceil \log \binom{u}{n} \rceil$ bits. To our knowledge, this result is the first of its kind. Even when considered from a worst-case perspective, our data structures are the first to take $O(n \log(u/n))$ bits with near-optimal query time. We also develop an indexable dictionary (ID) – a data structure supporting partial rank and select queries – in the same number of bits that supports each query even faster in $O(\log\log n)$ time. This result is the first to operate with gap-style bounds in space with time sublogarithmic in terms of the number of items stored. Moreover, our data structures are useful in practice; we also have a practical implementation and we discuss algorithmic engineering and experimental results in this paper. Our results show that *gap* is about 10%–40% of $\lceil \log \binom{u}{n} \rceil$ for many practical data sets.

1.1. Comparisons to previous work

Previous results of Jacobson [13], Munro [16], Brodnik and Munro [5], Pagh [17], and Raman et al. [19] develop dictionaries that support constant-time queries. The best among these are the ID (supporting partial rank and select) and the FID (supporting rank and select) by Raman et al. [19], which both support constant-time queries, and respectively require only $\lceil \log \binom{u}{n} \rceil + o(n) + O(\log\log u)$ bits and $\lceil \log \binom{u}{n} \rceil + O(u \log\log u/\log u) + O(\log\log u)$ bits. These results seem quite strong, as the constant factor associated with the information-theoretic minimum term is 1; unfortunately, the space is not bounded in a data-aware manner.

Recent work by Mäkinen and Navarro [15] and Sadakane and Grossi [20] achieves an FID with constant-time queries requiring $gap + O(n \log\log(u/n)) + O(u \log\log u/\log u)$ bits of space.[2] Both of these data structures are

[1] Throughout the paper, we assume the base of the logarithm is 2.

[2] The middle term $O(n \log\log(u/n))$ comes from encoding the extra bits needed for a prefix code (such as a δ code).

meaningful as methods to achieve constant-time queries over a *gap* representation. Still, these FID structures do not work well when $n \ll u$, as the $o(u)$ term will be much (even exponentially) larger than the information-theoretic minimum term $\lceil \log \binom{u}{n} \rceil$, dwarfing any savings we want to achieve. For instance, consider a typical example of maintaining a dictionary for IP lookup, storing say 2^{17} IP addresses out of a universe of size 2^{32}. In this case, $\lceil \log \binom{u}{n} \rceil$ is roughly 345,661 (about 2^{18}) bits while their $o(u)$ term is roughly 6.71×10^8 (about 2^{29}) bits – several orders of magnitude larger than the information-theoretic minimum $\lceil \log \binom{u}{n} \rceil$ bits.

Blandford and Blelloch [2] proposed an interesting scheme that allows easy transformation of any FID implemented with $O(n)$ pointers into another that requires $O(gap) + O(u^\alpha \log u)$ bits for any $0 < \alpha < 1$.[3] After the transformation, query time is slowed down by a factor of $1/\alpha$ compared with time required by the original dictionary. Blandford and Blelloch's scheme allows us to have FIDs with space bounded in a data-aware manner. However, their analysis still has a potentially excessive $u^{\Omega(1)}$ term. We note that their method can be tuned by some of the techniques developed in our paper to achieve $(1+\epsilon)gap$ bits of space. However, this increases their search time by a multiplicative factor of $1/\epsilon$. In addition, they require either complex RAM operations or a decoding table that may require more space. This is in part because their space-savings approach is fundamentally different from our own; it packs a variable number of items into a constant number of memory words and fetches the information in a constant number of RAM operations or by use of a large decoding table. In contrast, our data structure fetches one item at a time. We describe this structure in more detail in Section 4.

A fundamental aspect of a dictionary's search capabilities is captured by the predecessor problem, since dictionaries that (implicitly) solve the predecessor problem require fundamentally more space and time than those that do not. Precisely, the predecessor query determines the largest item in S smaller than the query. Fredman and Willard [9] proposed the well-known fusion tree which supports predecessor queries in $O(\log n / \log \log n)$ time. The query time was later improved by Beame and Fich's key result [4]. In particular, Beame and Fich describe a data structure that takes $O(n^2 \log u)$ bits of space so that membership and predecessor queries can be solved in $BF(u,n) = O(\min\{(\log \log u)/(\log \log \log u), \sqrt{(\log n)/(\log \log n)}\})$ time. They also show that this bound is tight as long as we have only $O(n^{O(1)} \log u)$ bits available.[4] Recently, Pătraşcu and Thorup [18] improved their space to $O(n^{1+\exp(-\log^{1-\epsilon} \log u)} \log u)$ bits of space, but unfortunately this improvement does not help our data structure.

Andersson and Thorup [1] provide a transformation to Beame and Fich's data structure, improving the space to $O(n \log u)$ bits and making the data structure dynamic using exponential search trees. However, the query time increases to

$$AT(u,n) = O\left(\min\left\{\sqrt{\frac{\log n}{\log \log n}}, \frac{\log \log u}{\log \log \log u} \cdot \log \log n, \log \log n + \frac{\log n}{\log \log u}\right\}\right).$$

Since rank and select can be used to answer predecessor queries, we improve Andersson and Thorup's structure in terms of space without sacrificing query time. In the worst case, our fully-indexable dictionary compares favorably with both Raman et al. [19] and Blandford and Blelloch [2]. With respect to the former, though we cannot support $O(1)$-time queries, we have eliminated the problematic $o(u)$ space term. Our query time – which is $AT(u,n)$ – is already close to the optimal $BF(u,n)$. For our indexable dictionary, when compared with Raman et al.'s ID structure [19], we pay a small price in the lookup time in exchange for achieving space bounds in terms of *gap*, which may be significant in practice.

The table in Fig. 1 lists the theoretical results with practical estimates for the space required to represent the various compressed dictionaries we mentioned. In all reported bounds, we refer to *fully-indexable dictionaries* (FID). Note that $BF(u,n) \leq AT(u,n)$ for any u and n.

1.2. Outline of the paper

The organization of the paper is as follows. In Section 2, we introduce three space measures for set data and show the strong relationship among them. In Section 3, we develop a binary-searchable dictionary representation (BSD),

[3] They only claim $O(n \log((u+n)/n)) + O(u^\alpha \log u)$ bits in their paper.

[4] It is this result which necessitates Raman et al.'s FID [19] $o(u)$ space term, since constant-time rank and select queries imply constant-time predecessor queries as well.

	Theoretical		Practical[a]
Paper	Time	Space (bits)	Space (bits)
this paper	$AT(u,n)$	$gap + o(\log \binom{u}{n})$ when $n \ll u$	$\leq 1,830,959$
[2]	$AT(u,n)$	$2gap + \Theta(u^{\epsilon})$	$\leq 1,855,116$
[21][b]	$O(\log\log u)$	$\Theta(n \log u)$	$> 3,200,000$
[1]	$AT(u,n)$	$\Theta(n \log u)$	$> 3,200,000$
[4]	$BF(u,n)$	$\Theta(n^2 \log u)$	$> 320,000,000,000$
[18]	$BF(u,n)$	$\Theta(n^{1+\exp(-\log^{1-\epsilon}\log u)} \log u)$	$> 10,000,000$
[13]	$O(1)$	$u + \Theta(u \log\log u / \log u)$	$> 4,429,185,024$
[19]	$O(1)$	$\log \binom{u}{n} + \Theta(u \log\log u / \log u)$	$> 136,217,728$
[15]	$O(1)$	$gap + O(n \log\log(u/n)) + \Theta(u \log\log u / \log u)$	$> 136,017,728$
[20]	$O(1)$	$gap + O(n \log\log(u/n)) + \Theta(u \log\log u / \log u)$	$> 136,017,728$

[a] The practical space bounds are for indexing our `upc_32` file, with $n = 100{,}000$ and $u = 2^{32}$. The values for [21,4,13,19,15,20] are estimated by their reported space bounds. For these methods, we relaxed their query times to $O(\log\log u)$ to provide a fairer comparison in space usage.

[b] The theoretical space bound is from Willard's y-fast trie implementation [22].

Fig. 1. Time and space bounds of dictionaries for *rank* and *select* queries.

which serves as an important component in our main results. In Section 4, we describe our fully-indexable dictionary and analyze it for both gap-style bounds and worst-case bounds. We achieve a fully-indexable dictionary supporting rank in $AT(u,n)$ time and select in $O(\log\log n)$ time, taking $gap + o(n \log(u/n))$ bits of space, or $O(n \log(u/n))$ bits in the worst case. Note that fully-indexable dictionaries that take $O(n^{O(1)} \log u)$ bits of space are subject to the lower bound of [4]; hence, these times are near-optimal with respect to $BF(u,n)$. In Section 5, we present our indexable dictionary result, which cannot solve predecessor queries, and can thus improve upon the query times from [4]. Section 6 details our experimental findings. We conclude in Section 7.

2. Dictionaries and data-aware measures

Let $S = \langle s_1, \ldots, s_n \rangle$ be an ordered set of n items from a universe $U = \{0, 1, \ldots, u-1\}$ of size u; that is, $i < j$ implies $s_i < s_j$. We want to represent S in a succinct form so that we can perform basic dictionary queries on its compressed representation. We define dictionaries more formally in Section 2.1. The normal concern of a dictionary is how fast one can answer a query, but space usage is also an important consideration. We would like the dictionary to use the minimum space for representing S, regardless of how quickly it can be searched. There are some common measures to describe this minimum space. The first measure is $n \log u$, which is the number of bits needed to store the items s_i explicitly in an array. The second measure is the information-theoretic minimum $\lceil \log \binom{u}{n} \rceil \approx n \log(u/n)$, which is the worst-case number of bits required to differentiate between any two distinct n-item subsets of universe U. In Section 2.2 we describe two more measures for representing the set S, motivating these as reasonable measures for analyzing the space required by a dictionary. We show strong relationships between these measures in Section 2.3, along with some experimental results that illustrate their relative performance.

2.1. The dictionary problem

The *dictionary* problem appears as a fundamental black box component in a number of applications used to offer fast access (for some queries, even constant-time access) to the data. Some examples include suffix arrays and IP lookup tries. Our interest is to exploit the great potential for a functional but compressed dictionary data structure. In some applications, dictionaries are the bottlenecks, both in terms of space and query time.

We describe some fundamental queries on set data. Here, $a \in U$. The $member(S,a)$ function indicates whether a appears in the set S. The $rank(S,a)$ function returns the number of items in S that are less than or equal to a. The $select(S,i)$ function returns the ith smallest item of S, for i ranging from 1 to n. The $prank(S,a)$ function is a rank function, but only for items of S. The $pred(S,a)$ function returns the predecessor of a, the largest item x in S such

that $x < a$. We define these formally below.

$$rank(S, a) = \big|\{s_i | s_i \leq a\}\big| \qquad member(S, a) = 1 \text{ if } a \in S, 0 \text{ otherwise}$$
$$select(S, i) = s_i \qquad prank(S, a) = rank(S, a) \text{ if } a \in S, -1 \text{ otherwise}$$
$$pred(S, a) = \max\{s_i | s_i < a\} \text{ if } rank(S, a-1) > 0, -1 \text{ otherwise.}$$

Jacobson [13] has discussed and motivated the power of *rank* and *select* functions at some length. In particular, he shows that the operation set {*rank*, *select*} can perform more powerful queries than the operation set {*member*, *pred*}. As a result, much of the subsequent work has considered *rank* and *select* as fundamental operations on dictionary structures (such as [19,17,2]). To further illustrate this point, note that the right-hand column can be defined solely in terms of *rank* and *select*. For instance, $member(S, a) = rank(S, a) - rank(S, a-1)$ and $pred(S, a) = select(S, rank(S, a-1))$ if $rank(S, a-1) > 0$. We now define some convenient notation to describe different kinds of dictionaries.

Definition 1. An *indexable dictionary* (ID) represents a subset $S \subseteq U$ and supports the queries $prank(S, a)$ and $select(S, i)$. A *fully-indexable dictionary* (FID) represents a subset $S \subseteq U$ and supports the queries $rank(S, a)$ and $select(S, i)$.

Fully-indexable dictionaries can solve predecessor queries, and so they immediately find application in rich problem areas as IP lookup structures [7], compressed text indexing [10], and suffix arrays [12].

Suppose that for the set S of n items, each item s_i is also associated with a piece of satellite data d_i. To allow quick retrieval of the satellite data once the item is given, we could consider a set S' of tuples of the form $\langle key, data\rangle$, with $S' = \{\langle s_1, d_1\rangle, \langle s_2, d_2\rangle, \ldots, \langle s_n, d_n\rangle\}$, and build a dictionary on S'. In this context, we define $lookup(S', a) = d_j$ when $a = s_j$ for some j and `null` otherwise.

Definition 2. A *lookup dictionary* (LD) is a data structure representing a set S' that supports the query $lookup(S', a)$.

Let $A = d_1 d_2 \ldots d_n$ be a bitvector of length $|A| = \sum_i |d_i|$ with the data d_i concatenated together. If each piece of satellite data d_i is of a fixed length r, a simple array structure of $n \times r$ bits can be used to store the satellite data. We can construct an ID on S, so that for any item s_i, the *prank* query returns the position in A where its satellite data is stored. Combining this with RRR's ID result, we obtain the following lemma, which is used extensively in our data structures in Sections 4 and 5.

Lemma 1. *There exists a lookup dictionary (LD) with $m(q + r)$ bits supporting $lookup(S', a)$ in constant time, where $m = |S'|$, $q \leq \log u$ is the number of bits to represent each key in S', and r is the number of bits for each satellite data.* □

When the satellite data are variable-length, we still store them using $\sum_i |d_i|$ bits. However, we need to know the starting position of each satellite data item. To do this, we store an ID on m items, where the ith item denotes the starting bit position of the ith piece of satellite data among the $\sum_i |d_i|$ possible positions. We ask *select* queries to determine the location of the ith satellite data item. The result of Blandford and Blelloch [3] on arrays of variable-length bitstrings also provides this functionality.

2.2. The gap and trie measures

One well-known method for representing the set S is *gap encoding* [6], which is often used in compressing inverted indexes. (We refer the reader to [23] for a detailed treatment of the various applications of this method, as well as a source for further references.) Consider the gaps between consecutive items in S, where the ith gap g_i is equal to $s_i - s_{i-1}$. We can now represent the set S as the stream of gaps $G = g_1, \ldots, g_n$, where $g_1 = s_1$, along with the value n. The stream G of gaps can be stored using variable-length encoding depending upon their size. Suppose we could store each g_i in $\lceil \log(g_i + 1)\rceil$ bits. Then, the total space, which we call the *gap measure*, is

$$gap(S) = \sum_{i=1}^{n} \lceil \log(g_i + 1)\rceil$$

bits. Note that we cannot merely store each g_i in $\lceil \log(g_i + 1)\rceil$ bits and decode the stream uniquely; we also need to know the separation boundaries between successive items. One popular technique to "mark" these separations is by

using a prefix code such as the δ code [8]. In δ coding, we represent each g_i in $\lceil \log(g_i + 1) \rceil + 2 \lceil \log \log(g_i + 1) \rceil$ bits, where the first $\lceil \log \log(g_i + 1) \rceil$ bits store the unary encoding of the number $\lceil \log \log(g_i + 1) \rceil$, the next $\lceil \log \log(g_i + 1) \rceil$ bits are the binary representation of the number $\lceil \log(g_i + 1) \rceil$, and the final $\lceil \log(g_i + 1) \rceil$ bits are the binary representation of g_i. We can then represent the stream of gaps $G = g_1, g_2, \ldots, g_n$ by concatenating the encoding of each g_i such that G is uniquely decodable. We refer to these extra bits of overhead beyond $gap(S)$ as the decoding overhead $Z(S)$. For δ coding, $Z(S) = 2 \sum_i \lceil \log \log(g_i + 1) \rceil$ bits. Our theoretical results in this paper make use of the δ code.

Another example of a prefix code is the nibble code proposed in [2]. In this paper, we will primarily use a variation of the nibble code called *nibble4* in our experiments. For this scheme, we write a "nibble" part of $\lceil \lceil \log(g_i + 1) \rceil / 4 \rceil$ in unary, which is followed by $4 \cdot \lfloor \lceil \log(g_i + 1) + 3 \rceil / 4 \rfloor$ bits to write the binary representation of g_i, padded out to multiples of four bits. (Later, we describe *nibble4fixed*, which we use for 64-bit data. It encodes the first part in binary in four bits, since for a universe size of 2^{64}, we would need to write $64/4 = 16$ different lengths.)

By Jensen's inequality,[5] $gap(S)$ is maximized when all gaps g_i are the same. In this case, $gap(S)$ would require roughly $n \log(u/n)$ bits, since each of the n gaps would be of size u/n. $Z(S)$ is also maximized in this case for δ coding. Hence, $Z(S)$ is roughly $2n \log \log(u/n)$ bits. Other prefix codes, such as the γ code [8] and some combination of Huffman and fixed-length coding, result in a somewhat different $Z(S)$. In this paper, we use the δ encoding scheme and denote the bit representation of S using this encoding by $\texttt{GAP}(S)$. The size of $\texttt{GAP}(S)$ is $|\texttt{GAP}(S)| = gap(S) + Z(S)$ bits.

Another method for compression of S is the *prefix omission method* (POM) [14], which is generally used to represent bitstrings of arbitrary length. Consider the bitstrings sorted lexicographically. We can represent each bitstring with respect to the previous bitstring by omitting the common prefix of the two. To compress S by POM, we think of each item of S as its $\log u$-length bit representation. The POM for S can also be seen as a subtree (of n leaves) of the complete binary tree on u leaves (which is a trie). We denote this subtree by $Tree(S)$. Each left edge of $Tree(S)$ represents a **0**, and each right edge represents a **1**. Each root-to-leaf path in this trie defines an item s in S.

For $x, y \in S$, let $x \ominus y$ denote the bitstring formed by omitting the common prefix of x and y from the bit representation of x. More precisely, let $|lcp(x, y)|$ denote the length of the longest common prefix of x and y; then, $x \ominus y$ is the last $\log u - |lcp(x, y)|$ bits of x. To represent S by POM, we generate the stream $L = l_1, l_2, \ldots, l_n$, where l_1 is the bit representation of s_1 in $\log u$ bits and $l_i = s_i \ominus s_{i-1}$. Let $|l_i|$ denote the number of bits in l_i. Thus, the cost of this representation, which we call the *trie measure*, is

$$trie(S) = \sum_{i=1}^{n} |l_i| = |s_1| + \sum_{i=2}^{n} |s_i \ominus s_{i-1}|,$$

which equals the number of edges in $Tree(S)$. Similar to the gap measure, the above representation with $trie(S)$ bits is not decodable as each string l_i is of variable length. Hence, we need some extra bits $Z'(S)$ for decoding, which takes $2 \sum_i \lceil \log |l_i| \rceil$ bits in the case of δ encoding. We use $\texttt{TRIE}(S)$ to denote the bit representation of S using POM, which takes $|\texttt{TRIE}(S)| = trie(S) + Z'(S)$ bits of space.

Let $S + a$ denote the set in which the positive integer a is added (modulo u) to each item of S. Thus, the set $S + a$ is $\{(s_1 + a) \mod u, (s_2 + a) \mod u, \ldots, (s_t + a) \mod u\}$. We define the *shifted trie measure* $strie(S) = \min_a \{trie(S + a)\}$, which corresponds to the number of bits needed to compress S by POM under the 'best shift'. We denote $\texttt{STRIE}(S)$ to be the corresponding $\texttt{TRIE}(S + a)$, and we define the space requirement $|\texttt{STRIE}(S)|$ similarly. Note that $|\texttt{STRIE}(S)|$ also includes the additional overhead of $\log u$ bits to store the number a to retrieve the original S. Next, we argue that $trie(S)$ could be somewhat larger than $gap(S)$, but $strie(S)$ is close to $gap(S)$.

Below, we summarize the notation introduced in this section.

$gap(S) = \sum_{i=1}^{n} \lceil \log(g_i + 1) \rceil$
$|\texttt{GAP}(s)| = gap(S) + Z(S)$
$trie(S) = |s_1| + \sum_{i=2}^{n} |s_i \ominus s_{i-1}|$
$|\texttt{TRIE}(s)| = trie(S) + Z'(S)$

$strie(S) = \min_a \{trie(S + a)\}$
$|\texttt{STRIE}(s)| = strie(S) + \log u$
$|x \ominus y| = \log u - |lcp(x, y)|$
$Tree(S)$ is a trie that stores the binary representations of items of S

[5] For a concave function f and $x_1 + x_2 + \cdots + x_k = x$, $\sum_i f(x_i)$ is maximized when $x_i = x/k$.

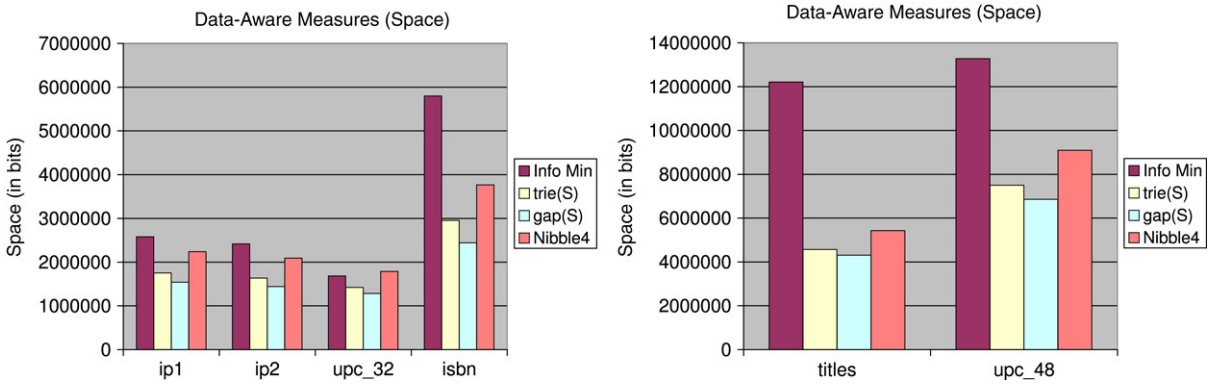

Fig. 2. Comparison of $\log \binom{u}{n}$, *trie*(S), *gap*(S), and a gap stream encoded according to the nibble4 code for the data files in Section 6.1.

2.3. Relationship between gap, trie and strie

In this section, we show a strong relationship between the *gap*, *trie* and *strie* measures. For any item s_i, $\lceil \log(g_i+1) \rceil$ is always smaller than $|l_i|$, but $|l_i|$ could be much larger. For example, when $s_{i-1} = 2^k - 1$ and $s_i = 2^k$, $|l_i| = k$ even though $\lceil \log(g_i + 1) \rceil = 1$. We show that this case cannot occur too frequently and prove that *trie*$(S) \leq 2$*gap*(S); furthermore, by applying a 'random shift', such cases are almost all eliminated. In the following lemma, we show that *trie*(S) can be more tightly bounded using this intuition.

Lemma 2. *The trie measure on the set $S + a$ requires trie$(S + a) \leq$ gap$(S) + 2n - 2$ bits on average over all values of $a \in [1, u]$.*

Proof. We proceed by showing that the sum $\sum_a$ *trie*$(S + a)$ is at most $u($*gap*$(S) + 2n - 2)$ bits. Recall that for a gap g_i, $|l_i|$ must be at least $\lceil \log(g_i + 1) \rceil$ bits long. For an arbitrary choice of a, $|l_i|$ can range from $\lceil \log(g_i + 1) \rceil$ to $\log u$ bits in length. We count how many times each $|l_i|$ contributes to the sum. For an arbitrarily chosen gap g_i, there are exactly g_i values of a such that $|l_i|$ will branch from *root*(*Tree*(S)). Thus, the total cost incurred is $g_i \log u$ bits. Similarly, there are $2g_i$ values of a such that $|l_i|$ would contribute $\log u - 1$ bits to the sum. In general, for $j < \log u - \lceil \log(g_i + 1) \rceil$, there are $2^j g_i$ values of a such that $|l_i|$ would contribute $\log u - j$ bits to the sum. Finally, the number of times $|l_i| = \lceil \log(g_j + 1) \rceil$ is at most $u(2^{\lceil \log(g_i+1) \rceil} - g_i)/2^{\lceil \log(g_i+1) \rceil}$. Thus, $|l_i|$ contributes to the sum with

$$\sum_{j=0}^{\log u - \lceil \log(g_i+1) \rceil - 1} 2^j g_i (\log u - j) + \frac{u\left(2^{\lceil \log(g_i+1) \rceil} - g_i\right)}{2^{\lceil \log(g_i+1) \rceil}} \lceil \log(g_i + 1) \rceil$$

$$= u \lceil \log(g_i + 1) \rceil - g_i \log u + \frac{2ug_i}{2^{\lceil \log(g_i+1) \rceil}} - 2g_i.$$

We also incur an additional cost associated with shifts such that $s_i + a > u$, where we charge $|l_i|$ with $\log u$ bits, contributing an additional $g_i \log u$ bits. Summing up and averaging over each of the u possible shifts, we see that the gap g_i requires an average of less than $\lceil \log(g_i + 1) \rceil + 2$ bits. We then sum this over all possible gaps, showing that an average *trie*$(S + a)$ is $\sum_{i=1}^{n} (\lceil \log(g_i + 1) \rceil + 2 - 2g_i/u) =$ *gap*$(S) + 2n - 2$ bits, thus proving the lemma. □

Since the minimum is less than the average, we obtain the following corollary.

Corollary 1. *The shifted trie measure, strie(S), is at most gap$(S) + 2n - 2$.*

Note that $|l_i|$ is bounded on average by $\lceil \log(g_i + 1) \rceil + 2$ bits. Since the decoding overhead is $\lceil 2 \log |l_i| \rceil$ with the δ code, we can bound the total overhead $2 \sum_i \lceil \log |l_i| \rceil$ by $2n \log \log(u/n)$ bits using Jensen's inequality. Thus, the space requirement $|\texttt{STRIE}(S)|$ is at most *strie*$(S) + 2n \log \log(u/n) + \log u$ bits.

We provide some experimental results on real data sets in Fig. 2, which bear out the theoretical findings in this section. Here, the files tested are described in Section 6.1, and the space is reported (in bits) along the y-axis. The figure on the left shows data files with a universe of size $u \leq 2^{32}$, and the figure on the right shows data files

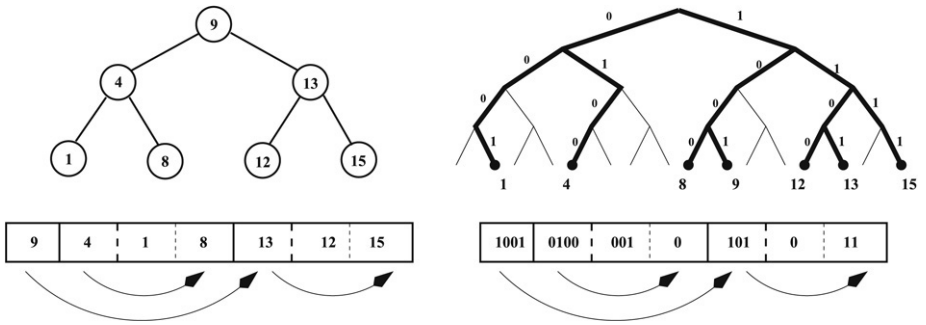

Fig. 3. The left-hand side shows a binary search tree built on the items 1, 4, 8, 9, 12, 13, and 15. Beneath that is its pre-order layout on disk, where the arrows represent pointers to the right subtree. The right-hand side shows the trie built on the same items. Beneath that is the corresponding layout on disk, but each item s is encoded with respect to $anc(s)$. For instance, 8 is encoded in the layout on the right as **0**, since $anc(8) = 9$ differs from it by a single bit.

with $u \leq 2^{64}$. Notice that $gap(S)$ is significantly smaller than $\log \binom{u}{n}$ for real data. In fact, nibble4 is a decodeable gap encoding that *also* outperforms the information-theoretic minimum. Since $gap(S)$ is less than $trie(S)$ for all of the files, we are free to use the gap measure for the remainder of our experimental results.

3. Binary-searchable dictionary representation

Despite all the development on the POM model, the trie encoding of S does not support time-efficient queries as we would like. Klein and Shapira [14] use the trie encoding to search in compressed dictionaries, but their searching algorithm essentially consists of a linear scan of the items in the dictionary and takes at least $\Omega(n)$ time. Most algorithms using gap encoding also need a linear scan. In this section, we build a binary-searchable data structure BSD, which resolves rank and select queries in $O(\log n)$ time. We show that the space required by this structure is *gap* bits plus low-order terms. In fact, the main point of this section is in showing that a binary-searchable representation requires about the same number of bits as simple linear encoding schemes. Also, BSD is our main building block and will be used later in this paper to support fast lookup in our FID and ID dictionary structures.

The BSD structure encodes a pre-order traversal of a balanced binary search tree T built on the n items of S. In Fig. 3, the pre-order traversal for the set S is 9, 4, 1, 8, 13, 12, and 15. The key point is that instead of storing each item s_i explicitly in $\log u$ bits, we encode an item with respect to an ancestor $anc(s_i)$, defined as follows. Let A_i be the set of all the ancestors of s_i in the binary search tree T. Then, $anc(s_i) = x \in A_i$ such that $lcp(s_i, x)$ is maximized over all ancestors in A_i. We represent s_i by $s_i \ominus anc(s_i)$ using $\log u - |lcp(s_i, anc(s_i))|$ bits, reminiscent of our trie encoding. Now we define the $\texttt{BSD}(S)$ encoding.

We use a recursive layout to describe the pre-order traversal of the binary search tree of n items. Let the subsets $S_L = \langle s_1, s_2, \ldots, s_{\lceil n/2 \rceil - 1} \rangle$ and $S_R = \langle s_{\lceil n/2 \rceil + 1}, \ldots, s_n \rangle$ represent the left and right subtrees of the $s_{\lceil n/2 \rceil}$th item. Generally, let $S_{i,j} = \langle s_i, s_{i+1}, \ldots, s_j \rangle$. Let $anc(s_{\lceil n/2 \rceil}) = 0$. For $\texttt{BSD}(S)$, let $|\texttt{BSD}(S)|$ denote the number of bits needed to encode $\texttt{BSD}(S)$. Then, we define the BSD encoding as

$$\texttt{BSD}(S) = \langle s_{\lceil n/2 \rceil} \ominus anc(s_{\lceil n/2 \rceil}); |\texttt{BSD}(S_L)|; \texttt{BSD}(S_L); \texttt{BSD}(S_R) \rangle.$$

Note that $s_{\lceil n/2 \rceil} \ominus anc(s_{\lceil n/2 \rceil})$ is a variable-length string, which is stored using δ coding. The term $|\texttt{BSD}(S_L)|$ constitutes additional overhead but is needed in order to jump to the right half of the set while searching. (We will call this term the *pointer cost*, and we will refer to it in our experimental section.) In fact, we could actually store just $\min\{|\texttt{BSD}(S_L)|, |\texttt{BSD}(S_R)|\}$ bits (with an additional n bits to indicate our choice), along with remembering whichever was smaller of the left and right subtrees.[6] Nevertheless, it turns out that BSD requires nearly the same space as does the TRIE encoding. Next, we describe how rank and select functions can be supported in $O(\log n)$ time using $\texttt{BSD}(S)$, and then we analyze the space usage of $\texttt{BSD}(S)$.

[6] Making this improvement would require the structure to be built from the bottom-up rather than with our recursive formulation above; we defer those details in the interest of clarity.

We use $\texttt{BSD}(S)$ as a black box on $O(\log n)$ items and achieve $O(\log\log n)$ time; however, in order to do so, we must be able to decode a δ-coded item (or bitstring) in $O(1)$ time in the RAM model. We assume that the word size of the machine is at least $\log u$ bits, and that we are allowed to perform addition, subtraction, multiplication, and bitshift operations in $O(1)$ time. We also assume that we can calculate the position of the leftmost **1** of a subword x of $\log\log u$ bits in $O(1)$ time. (This task is equivalent to calculating $\lceil\log(x+1)\rceil$ when the word x is seen as an integer.) We can also easily encode and decode the $\ominus$ operator using bitshifts and additions. These assumptions are sufficient to allow $O(1)$ decoding time. If this model is not applicable, we can simulate the decoding by explicitly storing the decoding result of every possible $\log\log u$-bit number in a table with $\log u$ entries. Note that this table takes $O(\log u \log\log\log u)$ bits, which is negligible overhead.[7]

In order to support *rank* and *select*, we just need to store the single value n (in $\log n$ bits) at the beginning of the BSD to indicate how many items are stored within the structure. Since our structure is a well-defined balanced binary tree, at any node x with n_x items, we know that the size of our left subtree contains $\lceil n_x/2\rceil - 1$ items, and our right subtree contains $n_x - \lceil n_x/2\rceil$ items. Hence, we can compute *rank* and *select* based upon this information. More precisely, given $\texttt{BSD}(S)$, $\mathit{rank}(S,a)$ and $\mathit{select}(S,i)$ can be computed in $O(\log n)$ time by calling the recursive functions $\mathit{rrank}(\texttt{BSD}(S),a,0,u,n)$ and $\mathit{rselect}(\texttt{BSD}(S),i,0,u,n)$ as detailed below. In the pseudocode, the function $\mathit{root}(B)$ returns the first encoded string in B (i.e., $\mathit{root}(B) = s_i \ominus \mathit{anc}(s_i)$), and the function $\mathit{decode}(x,\ell,r)$ returns the item s_i that corresponds to the root of B. The latter function can be computed by first determining $\mathit{anc}(s_i)$, which is one of ℓ or r based on the first bit of $\mathit{root}(B)$. Then, $s_i = (\mathit{anc}(s_i) \operatorname{div} 2^y) \times 2^y + \mathit{root}(B)$, where $y = |\mathit{root}(B)|$.

```
function rrank(B, a, ℓ, r, n) {
  if (n = 0) return 0;
  x ← root(B);
  z ← decode(x, ℓ, r);
  if (z = a) return ⌈n/2⌉ + 1;
  else if (z < a)
   return ⌈n/2⌉ +
     rrank(BSD(S_R), a, z, r, n − ⌈n/2⌉);
  else return rrank(BSD(S_L), a, ℓ, z, ⌈n/2⌉ − 1);
}
```

```
function rselect(B, i, ℓ, r, n) {
  x ← root(BSD(S));
  z ← decode(x, ℓ, r);
  if (i = ⌈n/2⌉) return z;
  else if (i > ⌈n/2⌉)
   return rselect(BSD(S_R), i − ⌈n/2⌉, z, r, n − ⌈n/2⌉);
  else return rselect(BSD(S_L), i, ℓ, z, ⌈n/2⌉ − 1);
}
```

We denote the $\mathit{rank}(S,a)$ and $\mathit{select}(S,i)$ that operate on $\texttt{BSD}(S)$ by $\mathit{BSD\_rank}(B,a)$ and $\mathit{BSD\_select}(B,i)$, where B is a pointer (of $\log u$ bits) to $\texttt{BSD}(S)$.

Lemma 3. *The* $\texttt{BSD}(S)$ *representation requires at most* $\mathit{trie}(S) + O(n\log\log(u/n))$ *bits and supports rank and select functions in* $O(\log n)$ *time.*

Proof. The space of $\texttt{BSD}(S)$ can be divided into three parts: (i) the space for all $s_i \ominus \mathit{anc}(s_i)$; (ii) their decoding overhead; and (iii) the space to encode all $|\texttt{BSD}(S_L)|$, used to jump to the right half of the encoding. We now describe the space required for each of these parts.

The space for (i) can be shown to be equal to the number of edges in $\mathit{Tree}(S)$, which is exactly $\mathit{trie}(S)$. To prove this, it suffices to show that each edge in $\mathit{Tree}(S)$ is encoded only once in its $\texttt{BSD}(S)$ representation. Let item s be encountered according to its pre-order binary search tree traversal. Let A be the set of all *ancestors* on the root-to-leaf path leading to s in the binary search tree. In the trie structure, the path to s must lay between two root-to-leaf paths in the trie: either the path leading to its rightmost encoded ancestor on its left l or its leftmost encoded ancestor on its right r. We encode $s \ominus \mathit{anc}(s)$, which must either be l or r. (This could be the parent of s.) Since no other edge in the trie that lies between the path to l and the path to r has been used thus far, each trie edge is encoded only once in any BSD structure.

For (ii), the overhead is analogous to $Z(S)$ and we can bound it by $O(n\log\log(u/n))$ using Jensen's inequality. In particular, we must encode the length of the new branch for s. Essentially, we are encoding n items out of a universe of *trie* bits to indicate the starting bit position of each branch's encoding. By Jensen's inequality, the worst case for this encoding occurs when all n items encode the length trie/n, requiring at most $n\log(\mathit{trie}/n) \le n\log((2\sum_i \log g_i)/n) = O(n\log\log(u/n))$ bits. We must also know $\mathit{anc}(s)$, the ancestor we chose to encode from.

[7] We could reduce the size of this table even further to $O(\log\log n\log\log\log\log n)$ bits by using a slightly different encoding scheme than the δ code.

We remember our choice automatically according to the first bit of the encoded string – a leading bit of **0** means we chose r and a leading bit of **1** means we chose l.

For (iii), we analyze this by considering the contribution of $|\mathtt{BSD}(S_L)|$ at each level of the binary search tree of S. At level 1, i.e. the root level, $|\mathtt{BSD}(S_L)|$ is at most $\log(n\log(u/n))$ bits. At level i, this contribution is maximized (by Jensen's inequality) when all of the 2^{i-1} contributing terms are equal. (In other words, all trees are the same size.) Thus, the space usage at level i is bounded by $2^{i-1}\log((n/2^{i-1})\log(u/n))$. Summing up, we have

$$\sum_{i=1}^{\log n} 2^{i-1}\log\left(\frac{n}{2^{i-1}}\log\frac{u}{n}\right) = O\left(n\log\log\frac{u}{n}+n\right),$$

which is a path recursion sum [11]. □

The above lemma suggests that $\mathtt{BSD}(S+a)$ would require fewer than than $trie(S+a)$ bits, plus $O(n\log\log(u/n))$ bits for any a. Thus by Corollary 1, $\min_a\{|\mathtt{BSD}(S+a)|\}$ is at most $gap(S)+O(n\log\log(u/n))$ bits. For the rest of the paper, we assume the BSD representation for S is based on its best possible shift. Thus, we obtain the following theorem, which will be used in further construction of our data structures in Sections 4 and 5.

Theorem 1 (*BSD*). *The representation* $\mathtt{BSD}(S)$ *is a fully-indexable dictionary (FID) occupying* $gap(S)+O(n\log\log(u/n))$ *bits while supporting rank and select functions in* $O(\log n)$ *time.* □

Next, we describe $\mathtt{BSGAP}(S)$, a simple and implementable variant of the $\mathtt{BSD}(S)$ representation that we use in our experimental results in Section 6. The key idea of $\mathtt{BSGAP}(S)$ is to directly encode the difference $|s_i - anc(s_i)|$ using gap encoding. Precisely, we replace the encoding $s_i \ominus anc(s_i)$ from $\mathtt{BSD}(S)$ by $\lceil\log(|s_i - anc(s_i)|+1)\rceil$. We also store one additional bit to indicate which ancestor encodes s_i. Using a similar analysis to that in Lemma 3, we arrive at the following corollary.

Corollary 2. *The representation* $\mathtt{BSGAP}(S)$ *is a fully-indexable dictionary (FID) occupying* $gap(S)+O(n\log\log(u/n))$ *bits while supporting rank and select functions in* $O(\log n)$ *time.* □

4. The fully-indexable dictionary structure

In this section, we describe our first main result, Theorem 2. We build a simple two-level hierarchical framework to obtain a fully-indexable dictionary (FID) such that *rank* takes $AT(u,n)$ time and *select* takes $O(\log\log n)$ time. The challenge in designing such a data structure lies in only spending $gap(S)+O(n\log(u/n)/\log n)+O(n\log\log(u/n))$ bits in the process.

We describe our structure in a bottom-up way. At the bottom level, we store a BSD dictionary for every $\lceil\log^2 n\rceil$ items from set S, each of which can resolve a rank or select query in $O(\log\log n)$ time. We also store $B.first\_rank$ along with each BSD B, where $B.first\_rank$ is the rank in S of its first item in B. We also keep an array $P[1..\lceil n/\log^2 n\rceil]$, where $P[i]$ stores a pointer to the ith BSD structure, which stores the items $s_{(i-1)\log^2 n+1},\ldots,s_{i\log^2 n}$. This structure alone is sufficient to support *select*. In order to support *rank*, let $\hat{S}=\{s_i|i \mod (\log^2 n)=1\}$ be the set of smallest items from each BSD. We build an instance of Andersson and Thorup's predecessor structure [1] on $\hat{S}$, called R. To support *rank*, we use a lookup dictionary L from Lemma 1 built on $\hat{S}$ as keys with pointers to the corresponding BSD as satellite data. We denote the process of looking up the satellite data associated with $s\in\hat{S}$ by $L.lookup(s)$. Then, *rank* and *select* can be solved as follows.

```
function rank(S, a) {
  s ← pred(R, a);
  B ← L.lookup(s);
  return B.first_rank + BSD_rank(B, a);
}
```

```
function select(S, i) {
  j ← ⌈i/(log² n)⌉;
  B ← P[j];
  return BSDselect(B, i − B.first_rank + 1);
}
```

We are almost ready to show the main theorem of this section, but first, we require the following lemma.

Lemma 4. *Let* $S_1,S_2,\ldots,S_k$ *be a partition of* S*, with each* S_i *consisting of items of consecutive ranks in* S*. Precisely, each* S_i *consists of items* $s_j,s_{j+1},\ldots,s_\ell$ *for some* $j\le\ell$*. Then,* $\sum_{i=1}^{k}|\mathtt{BSD}(S_i)|\le gap(S)+O(k\log u)+O(n\log\log(u/n))$.

Proof. Let $u_i = \max\{s \in S_i\} - \min\{s \in S_i\} + 1$ and $n_i = |S_i|$. By Theorem 1, $|\texttt{BSD}(S_i)| \leq gap(S_i) + O(n_i \log\log(u_i/n_i))$. Thus, the lemma follows since $\sum_{i=1}^{k} gap(S_i) \leq gap(S) + O(k \log u)$, and by Jensen's inequality, we can show that $\sum_{i=1}^{k} O(n_i \log\log(u_i/n_i)) \leq O(n \log\log(u/n))$. □

Based on the above lemma, we obtain the main theorem below, along with a worst-case analysis in Corollary 3, since *gap* and $O(n \log\log(u/n))$ are bounded by $O(n \log(u/n))$.

Theorem 2. *We implement a fully-indexable dictionary (FID) in* $gap(S) + O(n \log(u/n)/\log n) + O(n \log\log(u/n))$ *bits so that rank queries take* $AT(u, n)$ *time and select queries take* $O(\log\log n)$ *time.*

Proof. For *select*, we require $O(\log\log n)$ time to traverse the ith BSD dictionary. For *rank*, the time bound is dominated by the predecessor query in R, taking $AT(u, n/\log^2 n) = O(AT(u, n))$ time. This shows our time bounds. For our space bounds, the $n/\log^2 n$ BSD structures require a total of $gap(S) + O(n \log(u/n)/\log n) + O(n \log\log(u/n))$ bits. The array P and the field $B.first\_rank$ take at most $O(n/\log^2 n) \times \log u = O(n \log(u/n)/\log n)$ bits in total, proving the theorem. □

Corollary 3. *We implement a fully-indexable dictionary (FID) in at most* $O(n \log(u/n))$ *bits so that rank queries take* $AT(u, n)$ *time and select queries take* $O(\log\log n)$ *time.* □

Finally, we capture a technically interesting space–time tradeoff of our FID, obtained by scaling the size of the groupings. This observation implies that the second-order space term in our structure can be made arbitrarily small, at the cost of a slight increase in the query times.

Corollary 4. *For any* $\alpha > 1$, *we can implement a fully-indexable dictionary (FID) in total space* $gap(S) + O(n \log(u/n)/\log^{\alpha-1} n) + O(n \log\log(u/n))$ *bits so that the function rank takes* $AT(u, n/\log^{\alpha} n) + O(\alpha \log\log n)$ *time and the function select takes* $O(\alpha \log\log n)$ *time.* □

5. The indexable dictionary structure

In this section, we build upon the approach of the last section. We partition S into lower level BSD structures, each of size *at most* $\log^3 n$. We use a top level 'distributor' structure which enables us to access the correct BSD while answering a query. In contrast to the last section, if the query item is not present in S, our top level distributor may not return any associated BSD. Hence, we cannot support rank or predecessor queries.

Our top level distributor takes $O(\log\log n)$ time to return the correct BSD. This is less than $AT(u, n)$ time; the partitioning scheme is somewhat more complex than that in our FID. As a result, we can support partial rank or select queries in $O(\log\log n)$ time. To manage the space required, we limit the number of partitions to be at most $O(n \log\log n/\log^3 n)$, so that the overhead incurred by our top level distributor can be bounded by the same second-order term as in our FID.

Next, we describe our top level distributor structure, which is analogous to the van Emde Boas (VEB) tree [21]. With this distributor structure, on any given input x, we can report x is not in S, or obtain the BSD that can contain x efficiently.

5.1. The top level distributor structure

Our distributor structure is a recursive structure analogous to a VEB tree. Instead of having $O(\log\log u)$ levels of recursion as in the case for a VEB tree, our distributor has only $h = 3 \log\log n$ levels. At the top level (Level 1), we have a single distributor (with parameter $p = 0$ to be explained shortly) to distribute all items in S. For level $i = 1$ to $h - 1$, a Level i distributor with parameter p connects to some Level $i + 1$ distributors, which are then used to distribute the items recursively; the parameter p indicates that all the input items share the same first p bits. At the bottom level (Level h), a Level h distributor directs the items to their designated BSD structures. More precisely, for $i = 1$ to $h - 1$, a Level i distributor with parameter $p = p_i$ works as follows:

(1) Partition the items into groups according to the first $p_i + (\log u)/2^i$ bits.
(2) For each group with more than $\log^3 n$ items (which we call a dense group), the items are passed to a Level $i + 1$ distributor with parameter $p = p_i + (\log u)/2^i$.

(3) For all items not in a dense group, they are grouped together.
  (a) If the number of items is at most $\log^3 n$, the items are passed to a Level h distributor with parameter $p = p_i$.
  (b) Otherwise, the items are passed to a Level $i+1$ distributor with parameter $p = p_i$.

We can easily show the following by the recursive definition above: At a Level i distributor with parameter $p = p_i$, if we partition the items into groups based on the first $p_i + (2\log u)/2^i$ bits instead, the size of each group is at most $\log^3 n$. Making use of this fact, a Level h distributor with parameter $p = p_h$ partitions the n_h input items into groups based on their first $p_h + (2\log u)/2^h$ bits, such that each group is of size at most $\log^3 n$. The n_h items are then directed to the designated BSD data structures, with each BSD containing at most $O(\log^3 n)$ items. With the above data structure D, we can find the BSD that can contain x by calling *find_BSD*(D, x) as follows:

```
function find_BSD(D, x) {
  D_1 ← Level 1 distributor from D;
  i ← 1, p_1 ← 0;
  for i = 1 to h − 1
    (D_{i+1}, p_{i+1}) ← distribute(D_i, i, p_i, x);
    if (D_i is a Level h distributor) p_h ← p_i,
break;
  return retrieve_BSD(D_h, p_h, x);
}
```

```
function retrieve_BSD(D, p, x) {
  L ← the LD stored in D;
  y ← x[p + 1..p + (2 log u)/ log^3 n];
  return L.lookup(y);
}
```

The function *distribute*(D_i, i, p_i, x) retrieves the Level $i+1$ distributor with parameter $p = p_i$ in which x is distributed according to the first $p_i + (\log u)/2^i$ bits. The notation $x[\ell..r]$ $(\ell \le r)$ denotes the substring of the bitstring representation of x, starting at the ℓth bit and ending at the rth bit. The function $L.lookup(y)$ returns *lookup*$(S(L), y)$ if $y \in S(L)$, where $S(L)$ denotes the set of keys stored by L.

Once we obtain the BSD B that can contain x, determining whether x is in B can be done in $O(\log\log n)$ time. Thus, if *find_BSD*(D, x) can be done in $O(\log\log n)$ time, the total time to answer *member*(S, x) is also $O(\log\log n)$.

5.2. Distributor details

In this part, we give details of the distributor that supports *distribute*(D_i, i, p_i, x) at Level i $(i \in [1, h-1])$ and *retrieve_BSD*(D_h, p_h, x) at Level h efficiently. We make use of an LD of Lemma 1 to achieve this. Based on this implementation, we show that *find_BSD*(D, x) can be done in $O(\log\log n)$ time.

For $i = 1$ to $h-1$, a Level i distributor with parameter p maintains an LD of Lemma 1 that stores the $p + (\log u)/2^i$ bits corresponding to a dense group as keys, and storing the $\log u$-bit pointer to the corresponding Level $i+1$ distributor as satellite information. It also explicitly stores an 'escape' pointer to the Level h or the Level $i+1$ distributor that corresponds to items not in dense groups.

For a Level h distributor with parameter p, we use a different structure. Let n_h be the number of items managed by this distributor. We store the number k of distinct BSDs containing these n_h items and an array $A[1..k]$ storing the pointers to these BSDs. Recall that all the n_h items share the first p bits, and the distributor here distributes an item into a group according to the first $p + (2\log u)/2^h$ bits. Therefore, we maintain an LD of Lemma 1 for the $(2\log u)/2^h$ bits that corresponds to a non-empty group, starting at the $(p+1)$st position. For the satellite information, we store the array entry of the corresponding BSD, which again takes $(2\log u)/2^h$ bits.

5.2.1. A minor modification

If each BSD data structure corresponds to items in consecutive ranks, we can bound the total space by $gap + O(n\log\log(u/n))$ bits. Unfortunately, in the current scheme, a BSD data structure directed by a Level h distributor may not correspond to items of consecutive ranks. For instance, let s_i and s_j be two items in the same BSD; then at some level, an intermediate item s_{i+1} may be partitioned into a dense group, while s_i and s_j are items not in the dense group. Consequently, the intermediate item s_{i+1} is not stored in the same BSD as s_i and s_j.

In order to bound the space as desired, we use a little fix: for each existing BSD in the current scheme, we split the items into maximal groups of consecutive ranks, and store each group in a separate BSD. Essentially, we transform the existing BSD into a list of BSDs so that each new BSD corresponds to items of consecutive ranks. Then, a Level h distributor now directs the item into one of the k lists of BSDs (as opposed one of the k BSDs before). We store an array $A[1..k]$ for the pointers to the k lists; for each list, we store the number k' of BSDs it contains. (Note that $k' \le \log^3 n$, since the total number of items in a BSD is $O(\log^3 n)$.) We also store an array $B[1..k']$ such that $B[i]$ stores the pointer

to the BSD whose smallest item is the ith smallest among that of the other BSDs. With the above implementation, *distribute*(D, i, p, x) (Lines 3 and 6 in *find_BSD*(D, x)) for each $i = 1$ to $h - 1$ can be done in $O(1)$ time. Then at Level h, we obtain the list of BSDs that can contain x in $O(1)$ time. After that, we use binary search on x against the smallest items of the BSDs to find the BSD that can contain x (Line 8). The time required is $O(\log k')$, which is at most $O(\log \log n)$ since $k' \leq \log^3 n$. Then, *find_BSD*(D, x) can be done in $O(\log \log n)$ time.

5.3. Solving partial rank and select queries

The partial rank query can be readily supported by our data structure in $O(\log \log n)$ time, as shown in the pseudo-code below. To enable the select query, we additionally maintain an array $F[1..n/\log^3 n]$ such that $F[i]$ stores a pointer to a list of BSDs that can contain the items with rank in $[(i-1)\log^3 n, i \log^3 n]$. For each list, we store number k'' of BSDs in the list, and an array $G[1..k'']$ for pointers to the k'' BSDs such that $G[1].\mathit{first\_rank} < G[2].\mathit{first\_rank} < G[3].\mathit{first\_rank} < \cdots < G[k''].\mathit{first\_rank}$. Then, *select*$(S, j)$ can be solved in the following pseudo-code.

```
function prank(S, x) {
  B ← find_BSD(D, x);
  if (B = null) return -1;
  else
    r ← B.first_rank, r' ← BSD_rank(B, x);
    if ( BSDselect(r', B) = x ) return r + r' − 1;
    else return -1;
}
```

```
function select(S, j) {
  G ← F[⌈j/log^3 n⌉];
  k'' ← the number of BSDs in the list G;
  i ← BinarySearch(G, k'', j);
  r_i ← G[i].first_rank;
  return BSDselect(j − r_i + 1, G[i]);
}
```

The function *BinarySearch*(G, k'', j) returns i such that $G[i].\mathit{first\_rank} < j < G[i+1].\mathit{first\_rank}$ using binary search, which takes $O(\log k'')$ time. The total time required for *select* is $O(\log k'') + O(\log \log n) = O(\log \log n)$.

5.4. Space analysis

To bound the total space usage, we will make use of the following lemma.

Lemma 5. *We show that*

(1) *the total number of distributors,* $\sum_{i=1}^{h} d_i$*, is at most* $O(n \log \log n/\log^3 n)$*, and*
(2) *the total number of* BSD *data structures is at most* $O(n \log \log n/\log^3 n)$.

Proof. For all the distributors in our data structure, we use *Dist*(r, p, i) to denote the Level i distributor such that all the items managed by it share the same prefix r of length p. We call a distributor *dense* if it manages more than $\log^3 n$ items; otherwise, it is called *sparse*. Note that sparse distributors only occur at Level h.

For Level i, the number of dense distributors is at most $n/\log^3 n$, because the items they manage are disjoint. Thus, there are at most $3n \log \log n/\log^3 n$ dense distributors in total. For each sparse distributor *Dist*(r, p, h), there must exist a dense distributor *Dist*(r, p, i) for some i. We map *Dist*(r, p, h) to *Dist*(r, p, i) such that i is maximized. Note that it is a bijection. Thus, the number of sparse distributors is bounded by the number of dense distributors, and the first claim follows.

If two consecutive rank items s_j and s_{j+1} are stored in different BSDs, we call (s_j, s_{j+1}) a *cut*. A cut can happen in one of two ways: (1) if s_j and s_{j+1} come from two distributors, or (2) if s_j and s_{j+1} come from the same Level h distributor which is dense. Note that the number of cuts is equal to the number of BSDs. Now, we count the number of cuts as follows.

For cuts of the first type, consider the smallest level i such that the s_j and s_{j+1} are in different distributors, say D_ℓ and D_r. (This implies that they are at the same Level $i-1$ distributor.) Then, by the definition of a distributor, either D_ℓ or D_r must be dense. We associate the cut with the dense distributor(s). Then, in this mapping, a dense distributor can be associated with at most two cuts, namely when it takes the roles of D_ℓ and D_r, respectively. Thus, the number of cuts of the first type is bounded by the number of dense distributors, which is $O(n \log \log n/\log^3 n)$.

The number of cuts of the second type is, by definition, bounded by $O(n/\log^3 n)$. Thus, the second claim follows. □

Next, we notice that for a particular i, items managed by different Level i distributors are disjoint. Let d_i denote the number of Level i distributors in our data structure. Also, recall that the space for an LD is $O(m(q + r))$ bits where m is the number of items, q is the number of bits needed to represent each key (i.e., p_i bits for the LD in a Level i

distributor, and $2\log u/\log^3 n$ bits for the LD in a Level h distributor), and r is the number of bits for each satellite data (i.e., $\log u$ bits for the LD in a Level i distributor, and $2\log u/\log^3 n$ bits for the LD in a Level h distributor). Then, for any i in $[1, h-1]$, the space occupied by all Level i distributors is equal to the space of LD for dense groups + space for escape pointers $\leq O(n/\log^3 n \times \log u) + d_i \log u$ bits. On the other hand, the space occupied by all Level h distributors is equal to space of LD for non-empty groups + space for k + space for arrays $A[1..k]$ and k' + space for arrays $B[1..k'] \leq O(n \times 2\log u/\log^3 n) + d_h \log u + O(n/\log^3 n + d_h) \times \log u + O(n \log\log n/\log^3 n) \times \log u$ bits, where the inequality follows from Lemma 5.

Next, the extra space needed by the partial rank and select structures is equal to space for rank of the smallest item of each BSD+ space for $F[1..n/\log^3 n]$ and k'' + space for $G[1..k''] \leq O(n \log\log n/\log^3 n) \times \log u + O(n/\log^3 n) \times \log u + O(n \log\log u/\log^3 n) \times \log u$ bits. In total, the space requirement for all the distributors is at most $\sum_{i=1}^{h}(O(n/\log^3 n) + d_i)\log u + O(n(\log\log n)(\log u)/\log^3 n)$ which is equal to $O(n(\log\log n)(\log u)/\log^3 n) + \sum_{i=1}^{h} d_i \log u \leq O(n(\log\log n)(\log u)/\log^3 n)$ bits, where the last inequality is based on Lemma 5.

Finally, since the above space terms can be bounded by $O(n\log(u/n)/\log n)$ and the space of all the BSD data structures is bounded by $gap + O(n\log\log(u/n))$ bits (Lemma 4), we have the following theorem.

Theorem 3. *Given a set S of n items from a universe $[1, u]$, we implement an indexable dictionary (ID) in $gap(S) + O(n\log(u/n)/\log n) + O(n\log\log(u/n))$ bits supporting partial rank and select queries in $O(\log\log n)$ time.* □

6. Experimental results

In this section, we present our experimental results, based on the BSGAP structure from Corollary 2. Recall that the BSGAP structure is organized similarly to a BSD, but gap encodes the difference between an item s and its best ancestor $anc(s)$. Section 6.1 describes the experimental setup that we use for our results. In Section 6.2, we discuss various issues with the space requirements of our BSGAP structure and give some intuition about how to encode the various parts of the BSGAP structure efficiently. In Section 6.3, we describe a further tweakable parameter for our BSGAP structure and use it as a black box to succinctly encode blocks of data.

Apart from the δ code, the nibble code [2], and the nibble4 code we have mentioned in Section 2.1, in this section, we also refer to a number of variations of prefix codes as follows:

- The *delta squared code* encodes the value $\lceil\log(g_i + 1)\rceil$ using δ codes, followed by the binary representation of g_i. For instance, the delta squared code for 170 is **001 00 1000 10101010**.
- The *nibble4Gamma* encodes the "nibble" part of the nibble4 code using the γ code instead of unary.[8] For instance, the nibble4Gamma code for 170 is **01 0 10101010**.
- In case the universe size of the data set is at most 2^{32}, we will also have the *fixed5 code* which encodes the value $\lceil\log(g_i + 1)\rceil$ in binary using five bits. For instance, 170 is encoded as **01000 10101010**.
- For larger universe sizes (such as our 2^{64}-sized ones), we use the *nibble4fixed code*, a mix of the nibble4 code and the fixed5 code. Here, we encode the "nibble" part of the nibble4 code using four bits.

For each of these codes, we create a small table of values so that we can decode them quickly when appropriate. As described in Section 3, these tables add negligible space, and we have accounted for this (and other) table space in the experimental results that we describe throughout the paper.

6.1. Experimental setup

Our source code is written in C++ in an object-oriented style. The experiments were run on a Dell PowerEdge 650 with 3 GB of RAM. The machine was running Centos 4.1, with a gnu g++ 3.4.4 compiler. The data sets used were as follows:

- **ip1:** List of IP addresses obtained from Duke University's Computer Science Department. The list refers to 159,690 IP addresses that hit the Duke CS pages in the month of January 2005.

[8] The "nibble" part will be an integer between 1 and 16. The γ code for an integer x is a unary encoding of $\lceil\log x\rceil$ followed by the binary encoding of x in $\lceil\log(x + 1)\rceil$ bits.

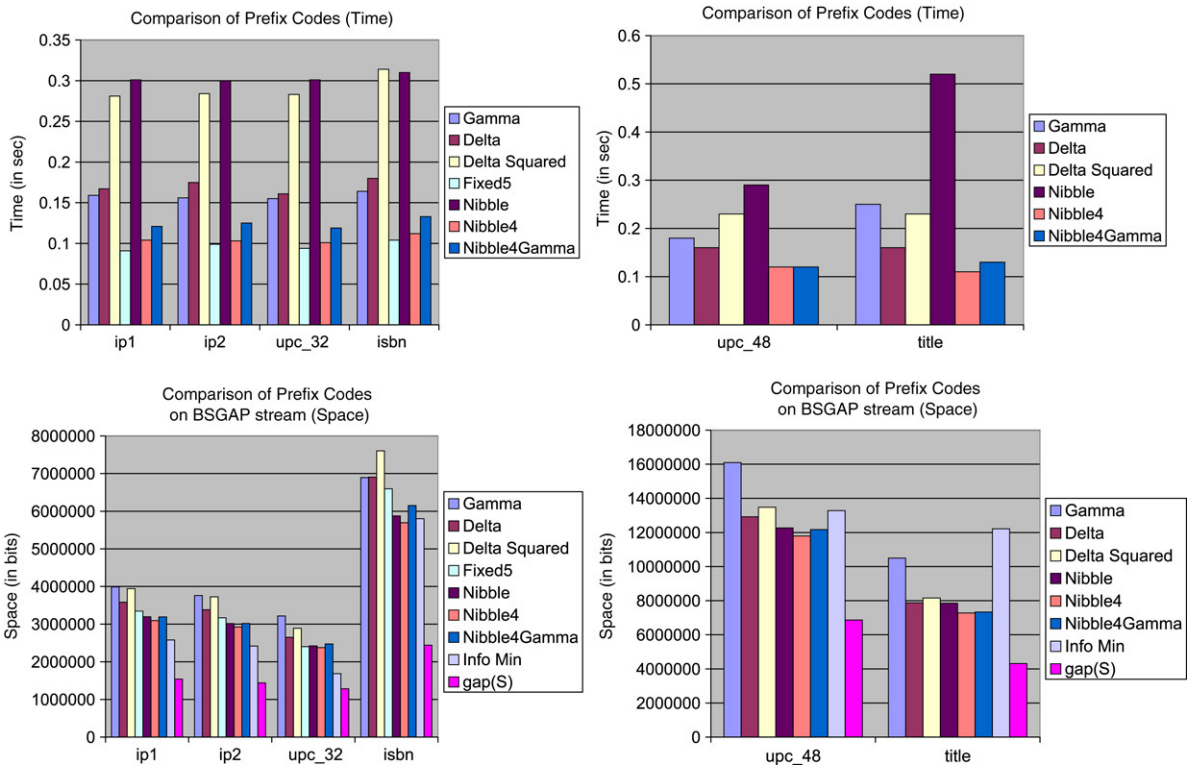

Fig. 4. Comparison of codes and measures for the data files in Section 6.1.

- `ip2`: Similar to `ip1`, but this list consists of 148,700 IP addresses that hit the Duke CS pages in February 2005.
- `upc_32`: List of 100,000 UPC codes obtained from items sold by the Wal-Mart supermarket that fit in a universe of size 2^{32}.
- `isbn`: List of 390,000 ISBNs of books at the Purdue Libraries in a 32-bit format.
- `upc_48`: List of 432,223 UPC codes in the original 48-bit format obtained from items sold by the Wal-Mart supermarket.
- `title`: List of 256,391 book titles from Purdue Libraries, converted into a numeric value out of a universe of size 2^{64}.

6.2. *Code comparisons for encodings and pointers*

We performed experiments to compare the space/time tradeoffs of using different encodings in place of nibble4. We summarize those experiments in Fig. 4. The figures in the top row show the time required to process 10,000 randomly generated *rank* queries with a BSGAP structure using the codes listed, averaged over 10 trials. The figures in the bottom row show the space (in bits) required to encode the BSGAP data structure using the listed prefix codes. Each of the bottom two graphs also has the information-theoretic minimum and $gap(S)$ listed for reference. Fig. 5 shows the same experiment for the raw gap stream; notice the minimal overhead incurred by using BSGAP.

It is clear that both fixed5 and nibble4 are very good codes in the BSGAP structure for the 32-bit case; fixed5 is slightly faster than nibble4, and nibble4 is slightly more space efficient. (For the `isbn` file, nibble4 is significantly more space efficient.) For 64-bit files, nibble4 is the clear choice. Since our focus is on space efficiency, the rest of the paper will build BSGAP structures with nibble4. (For our 64-bit data sets, we will actually use nibble4fixed.)

Next, we investigate the cost of these BSGAP pointers and see if a different choice of code *for just the pointers* can improve its cost. We summarize the space/time tradeoffs in Fig. 6. The figure shows the pointer costs (in bits) of each BSGAP structure. As we can see, nibble4 and nibble are both space efficient for the pointer distribution. However, nibble4 is again the logical choice, since it is both the most space efficient and very fast to decode. If we remove these pointer costs from the total space cost for the BSGAP structure, we see that this space is *about the same as encoding the gap stream sequentially*; as such, we can think of the pointer overhead for BSGAP as a cost to support fast searching.

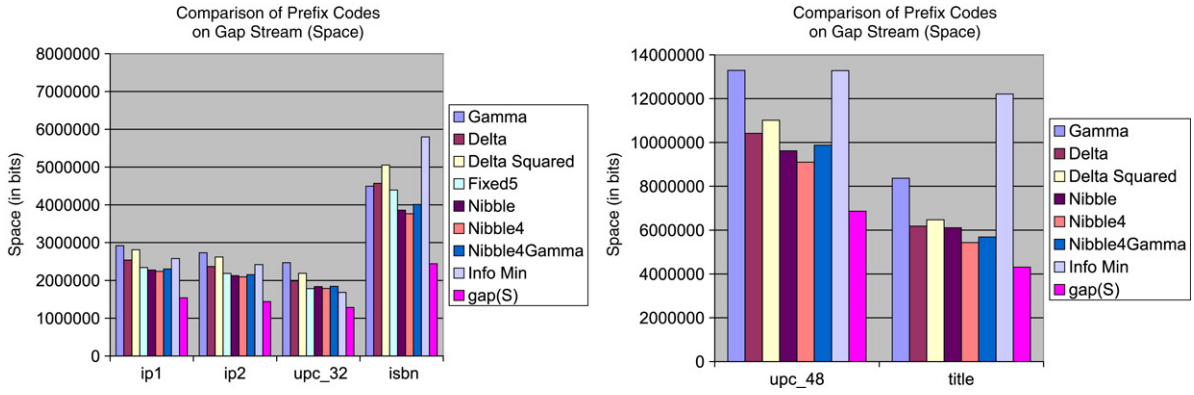

Fig. 5. Comparison of gap+codes, $\log \binom{u}{n}$, and $gap(S)$ for real data files, described in Section 6.1.

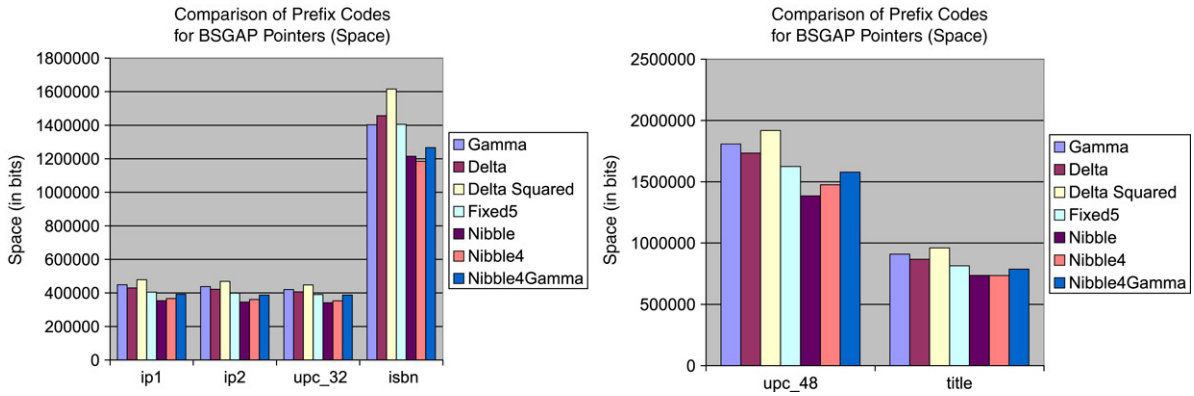

Fig. 6. Comparison of prefix codes for BSGAP pointers for the data files in Section 6.1.

6.3. BSGAP: *The succinct binary-searchable black box*

In this section, we focus on the practical implementation of our fully-indexable dictionary, modeled after Corollary 2. To make our practical dictionary, we replace [1] with a simple binary search tree, and introduce a new parameter $h = O(\log \log n)$ that does not affect the theoretical time for BSGAP but provides a noticeable improvement in practice. For each group of $\log^2 n$ items that is stored using BSGAP, we further tune our structure to resort to a simple sequential encoding scheme when there are at most h items left to search, where $h = O(\log \log n)$. Theoretically, the time required to search in the BSGAP structure is still $O(\log \log n)$. We employ this technique when sequential decoding is fast enough, to avoid writing bits to jump to the right half of the tree. (We call this the *pointer cost*.) In our experiments, we actually let h range up to $\log^2 n$, to see the point at which a sequential decoding of h items becomes impractical. It turns out that these few adjustments to our theoretical work result in a fast and succinct practical dictionary.

For the rest of the section, we define a parameter b that governs the number of items contained in each BSGAP structure and a parameter h that controls the degree of sequential encoding within a BSGAP data structure, as described above. We denote a particular configuration of our dictionary structure by $D(b, h)$. Let BB refer to the data structure in [2]. In this framework, BB is a special case of our dictionary $D(b, h)$ when $h = b$.

In Fig. 7, we show a space/time tradeoff for BB and our dictionary. Each graph plots space vs. time, where the time is that required to process 10,000 randomly generated *rank* queries, averaged over five trials. Here, we tune BB to operate on the same number of items in each block to avoid extra costs for padding and give them the same benefits as BSGAP receives. For each graph in Fig. 7, we let the blocksize b range from $[2, 256]$ and the hybrid value range from $[2, b]$. We collect time and space statistics for each $D(b, h)$ data structure. The BB curve is generated from the 256 points corresponding to $D(b, b)$. For the BSGAP curve, we partition the x-axis into 300 partitions and choose the most time-efficient implementation of $D(b, h)$ taking that much space. Notice that our BSGAP structure converges to BB as we allow more space for the data structures, but we have some improvement for extremely small space.

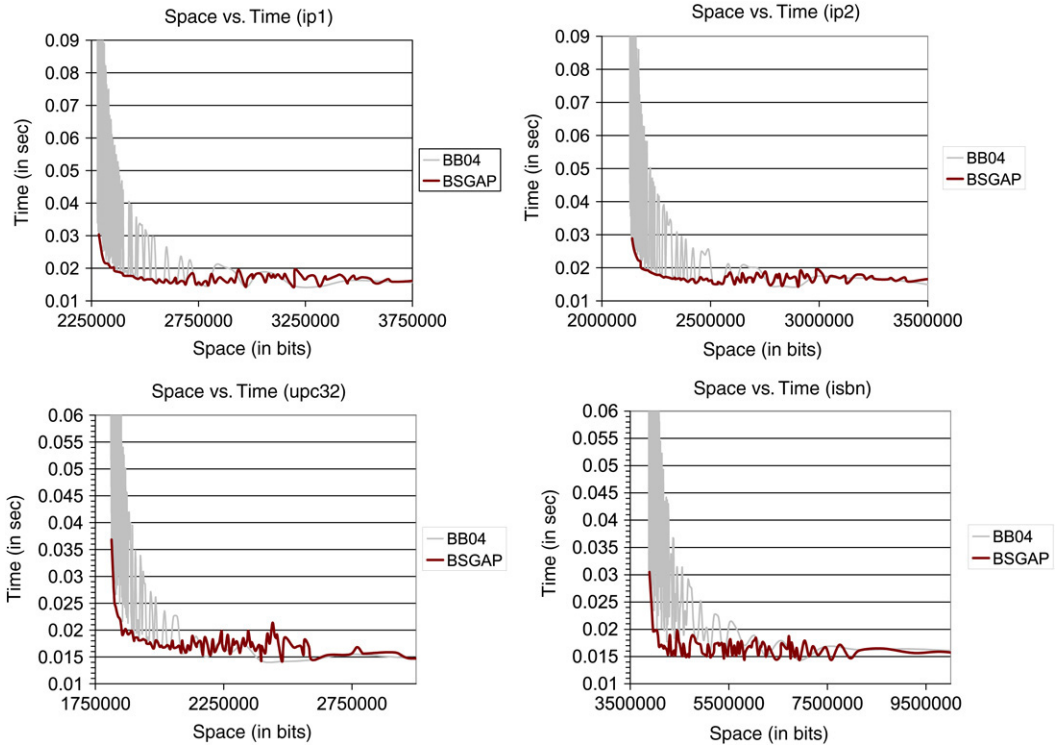

Fig. 7. Comparison of BB and BSGAP on 32-bit data files in Section 6.1.

Since BB is a subcase of our BSGAP structure, one might think that our space–time curve should never be higher than BB's. However, the curve is generated with actual data structures $D(b, h)$ taking a particular space and time. So, the existence of a point above the BB curve on our BSGAP curve simply means that there exists one configuration of our data structure $D(b, h)$ which has those particular results.

The parameter h is crucial to achieving a good space/time tradeoff. Notice that as h increases, the space of $D(b, h)$ decreases because we store fewer pointers in each BSGAP data structure. One may think of transferring this saved space into entries in the top level binary search tree to speed up the query time. On the other hand, the time required to search at the bottom of each BSGAP structure increases linearly with h. So, there must be some moderate value of h that balances these costs and arrives at the best space/time tradeoff. Hence, we collect all (b, h) pairs and evaluate the best candidates among them.

In Fig. 8, we compare BB and our dictionary for 64-bit data. We plot space vs. time, where the time is that required to process 1000 randomly generated *rank* queries, averaged over five trials. We collect data for $D(b, h)$ as before, where the range for b and h for `upc_48` is $[2, 512]$ and `title` is $[2, 2048]$. Notice that our data structure provides a clear advantage over BB as the universe size increases.

7. Conclusions

In this paper, we have formalized and developed measures for analyzing the space needed to store set data. These measures can provide a framework for further investigation of compressed data structuring techniques. We have achieved a fully-indexable dictionary that operates in near-optimal time ($AT(u, n)$) to support rank, select, and predecessor queries, while just taking $gap + O(n \log(u/n)/\log n) + O(n \log\log(u/n))$ bits of storage. This result improves a number of compressed data structures [19,1,2] by reducing space usage, while maintaining nearly optimal time bounds. Our *gap* term has a constant of 1, which is extremely important when considering matters of space efficiency. Equally important are the properties of the other space terms – if $n = o(u)$, they amount to $o\left(\log \binom{u}{n}\right)$ bits. Also, our dictionary is the first that achieves $O(n \log(u/n))$ bits of space, without significantly sacrificing the query times. (Recall that we take $AT(u, n) \geq BF(u, n)$ time.) We also provide an indexable dictionary which operates in $gap + O(n \log(u/n)/\log n) + O(n \log\log(u/n))$ bits and supports queries in $O(\log\log n)$ time. We conjecture that

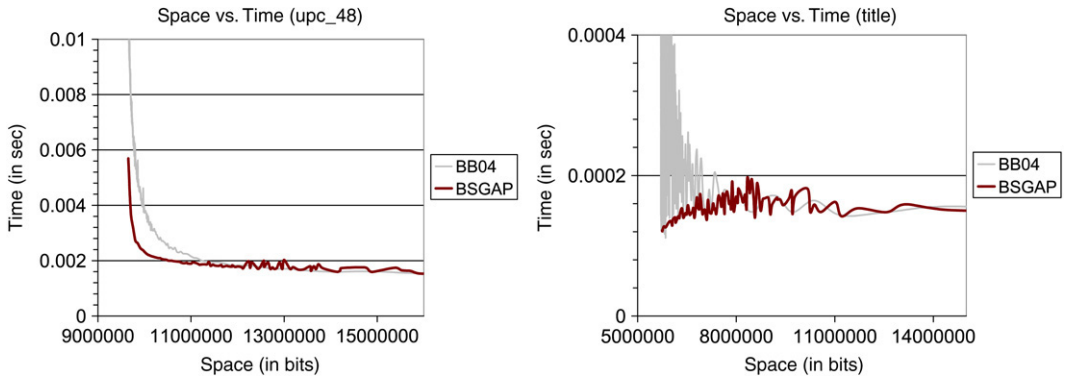

Fig. 8. Comparison of BB and BSGAP on 48-bit and 64-bit data files in Section 6.1.

if the space for an ID is measured in terms of *gap*, $O(1)$ query time may not be possible to achieve. Since the *gap* measure inherently exploits the encoding of items with respect to other items, $O(1)$ decoding time of an item (and thus searching) is not straightforward.

In addition, we have shown evidence that data-aware measures (such as *gap*) tend to be smaller than combinatorial measures on real-life data. Employing techniques that exploit the redundancy of the data can lead to more succinct data structures and a better understanding of the underlying information. As such, we encourage researchers to develop theoretical results with a data-aware analysis. In particular, our BSGAP data structure, along with BB (proposed in [2]) are extremely succinct in practice for sparse data sets. In addition, we provide some evidence that BSGAP is less sensitive than [2] to an increase in the size of the universe. Finally, we provide some useful information on the relative performance of prefix codes with respect to compression space and decompression time.

There are two open problems. Is it possible to give an indexable dictionary with query times further reduced, and with space measured in a data-aware manner? Another problem is whether we can extend our data structures to support dynamic operations.

Acknowledgments

We would like to thank Roberto Grossi for many fruitful discussions. The fourth author was supported in part by an IBM Faculty Research Grant.

References

[1] A. Andersson, M. Thorup, Tight(er) worst-case bounds on dynamic searching and priority queues, in: ACM Symposium on Theory of Computing, STOC, 2000.
[2] D. Blandford, G. Blelloch, Compact representations of ordered sets, in: Proceedings of the ACM–SIAM Symposium on Discrete Algorithms, January 2004.
[3] D. Blandford, G. Blelloch, Dictionaries using variable-length keys and data, with applications, in: Proceedings of the ACM–SIAM Symposium on Discrete Algorithms, January 2005.
[4] P. Beame, F. Fich, Optimal bounds for the predecessor problem, in: ACM Symposium on Theory of Computing, STOC, 1999, pp. 295–304.
[5] A. Brodnik, I. Munro, Membership in constant time and almost-minimum space, SIAM Journal on Computing 28 (5) (1999) 1627–1640.
[6] T.C. Bell, A. Moffat, C.G. Nevill-Manning, I.H. Witten, J. Zobel, Data compression in full-text retrieval systems, Journal of the American Society for Information Science 44 (9) (1993) 508–531.
[7] P. Crescenzi, L. Dardini, R. Grossi, IP address lookup made fast and simple, in: European Symposium on Algorithms, ESA, 1999, pp. 65–76.
[8] P. Elias, Universal codeword sets and representations of the integers, IEEE Transactions on Information Theory IT-21 (1975) 194–203.
[9] M.L. Fredman, D.E. Willard, Surpassing the information theoretic bound with fusion trees, Journal of Computer and System Sciences 47 (3) (1993) 424–436.
[10] R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, in: Proceedings of the ACM–SIAM Symposium on Discrete Algorithms, January 2003.
[11] D.H. Greene, D.E. Knuth, Mathematics for the Analysis of Algorithms, Birkhäuser, Boston, 1981.
[12] R. Grossi, J.S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, in: Proceedings of the ACM Symposium on Theory of Computing, vol. 32, 2000.
[13] G. Jacobson, Succinct static data structures, Technical Report CMU-CS-89-112, Dept. of Computer Science, Carnegie–Mellon University, January 1989.

[14] S.T. Klein, D. Shapira, Searching in compressed dictionaries, in: Data Compression Conference, DCC, 2002.
[15] V. Mäkinen, G. Navarro, Rank and select revisited and extended, Theoretical Computer Science (2006).
[16] J.I. Munro, Tables, Foundations of Software Technology and Theoretical Computer Science 16 (1996) 37–42.
[17] R. Pagh, Low redundancy in static dictionaries with $O(1)$ worst case lookup time, in: Proceedings of the International Colloquium on Automata, Languages, and Programming, in: Lecture Notes in Computer Science, vol. 1644, Springer-Verlag, 1999, pp. 595–604.
[18] M. Pătraşcu, M. Thorup, Time–space trade-offs for predecessor search, in: Proceedings of the ACM Symposium on Theory of Computing, 2006, pp. 232–240.
[19] R. Raman, V. Raman, S.S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, in: ACM–SIAM Symposium on Discrete Algorithms, 2002, pp. 233–242.
[20] K. Sadakane, R. Grossi, Squeezing succinct data structures into entropy bounds, in: ACM–SIAM Symposium on Discrete Algorithms, SODA, 2006, pp. 1230–1239.
[21] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Mathematical Systems Theory 10 (1977) 99–127.
[22] D.E. Willard, New trie data structures which support very fast search operations, Journal of Computer and System Sciences 28 (3) (1984) 379–394.
[23] I.H. Witten, A. Moffat, T.C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, second ed., Morgan Kaufmann Publishers, Los Altos, CA, 1999.